()

Generative CSG Models for Real Time Graphics

B. Guenter & M. Gavriliu

Abstract

Generative parametric CSG models, introduced by Snyder in 1992, have many desirable properties. Their functional representation is generally quite compact and resolution independent, since surfaces are represented as piecewise continuous functional programs. However, in practice they have proven impractical for real time rendering because of the difficulty of compactly and exactly representing the implicit curves of intersection between general parametric surfaces and because there was no published algorithm for triangulating the surfaces in real time. Our new algorithm computes an exact, piecewise parametric representation for the implicit curves of intersection. The new piecewise parametric curve representation is very compact and can be evaluated efficiently at run time, making it possible to change triangulation density dynamically. We have also developed a triangulation algorithm which effectively uses modern GPU's to render generative CSG models at high speed. Complex generative CSG models made with our system have a memory footprint of just 7-11 KBytes, which is orders of magnitude smaller than the equivalent polygonal mesh representation.

1. Introduction

Figure 1: *Procedural model of a speaker. Memory footprint of the object is approximately 8.9 KBytes. Rendering speed is 20 million triangles/sec.*

Generative parametric CSG models for 3D surfaces, first described in Snyder's seminal work [Sny92a], have many desirable characteristics for real time rendering. They are powerful enough to model many types of objects, especially manufactured things which frequently have simple procedural descriptions. The intrinsic generative model description is very compact and resolution independent because surfaces are represented as piecewise continuous functions.

Compactness is particularly important because of a long term trend in computer hardware: GPU processor performance has been increasing at a rate of roughly 71% per year while memory bandwidth has been increasing at only 25% per year [Owe05]. Fetching data from memory is steadily becoming more expensive relative to compututation. A more compact representation can be faster to render than a larger one, even if much more computation is required to process the compact representation. In terms of compactness generative models are a better fit to this trend than polygon meshes.

Compactness is also important for memory constrained platforms such as game consoles where large virtual worlds must be stored in RAM or streamed off of slow serial devices such as DVD drives. Bandwidth limited applications, such as online games, would also benefit from a more compact model representation.

In spite of these desirable qualities generative CSG models, introduced almost 15 years ago, are not used for real time rendering. Generative CSG modeling today is approximately where subdivision surfaces, introduced in 1978

[CC78, DS78], were 10 years ago. For a decade after their introduction they were a subject of much academic research but were hampered by various technical problems, such as the inability to easily specify creases. They were essentially unused commercially until approximately 10 years ago when solutions to some of these problems were found [DKT98, Sta98] and now they are ubiquitous.

Generative models have suffered from similar technical problems. Up to now it has not been possible to both render them quickly and retain their compact nature. For example, in Snyder's implementation of generative CSG models, till now the state of the art, the shape representation language was interpreted so evaluating points on the surface of the CSG object was far too slow for real time rendering.

A deeper problem was that the curve of intersection between the general parametric surfaces involved in the CSG operation was known only implicity. Triangulating the object required computing points on the implicit intersection curve. This required robust root finding techniques for general, not just algebraic, functions. Snyder used Interval Newton root finding. This was much too slow to be used at run time so curve points had to be pre-computed offline and the CSG model triangulated at some fixed resolution. As a consequence the resulting pre-triangulated generative CSG models were no longer resolution independent nor compact.

The primary contribution of this paper is a new algorithm for finding a piecewise parametric representation for the implicit curve of intersection between two general parametric surfaces involved in a CSG operation. This parametric representation is compact and exact to the limits of precision of floating point arithmetic. Arbitrary points on the intersection curve can be efficiently evaluated at run time which allows triangulation density to be adapted dynamically.

A secondary contribution is that we have developed an algorithm for efficiently triangulating generative CSG models in real time which effectively uses the capabilities of modern graphics hardware.

We have also developed a new language, called Shapes, for representing generative CSG models, and a compiler that transforms the high level Shapes programs to HLSL code which is executed directly on the GPU. The Shapes language is powerful enough to be used as the basis for a reasonably general purpose 3D modeling tool, much as the PostScript programming language can be used to represent 2D shapes for a 2D modeling program.

All of the objects illustrated in this paper are Shapes programs that were generated with a simple 3D modeling tool we created to demonstrate the potential of generative CSG modeling. Space limitations preclude a comprehensive discussion of the Shapes language, compiler, and symbolic differentiation system. These topics will be the subject of a separate paper. The important feature of Shapes to be aware of for the remainder of the paper is that generative CSG models are represented as Shapes programs which are compiled and then executed on the GPU in order to render the object.

We believe that our solutions to the problems of representing the implicit curve of intersection and triangulating generative CSG models in real time have finally made generative CSG models practical for real time rendering. Their many advantages should lead to widespread adoption.

2. Previous Work

Our work has some superficial similarities to previous work on real time CSG modeling but is distinguished by two characteristics. First, the class of parametric surfaces that can be represented in our system is quite general. The primary limitation is that the surfaces must be at least piecewise C^2 and must not be self intersecting. Some less important restrictions are noted in the body of the paper. Second, we compute an exact boundary surface representation of the CSG object which can be used for real time rendering. This is in contrast to the work of [BKZ01, PKKG03, AD03] where approximate intersections are computed and where surfaces must be represented by a specific function type.

Screen space techniques such as [SLJ03] can also be used to interactively compute CSG operations. These algorithms work at image space resolution and can can have $O(n^2)$ complexity in the worst case where n is the depth complexity at a pixel. These algorithms require many writes and reads to and from a Z-buffer. This type of memory intensive algorithm does not match well with long term hardware trends. In addition, Z-buffer precision problems can cause regions near the intersection of two objects to have a more jagged appearance than they would if the boundary of the surface was explicitly represented.

Our exact boundary surface representation ensures that only the parts of the object that result from the CSG operation will be rendered. No unnecessary writes will be made to the frame buffer, and no reads from the frame buffer are required.

3. Creating and Rendering a Shapes Object

Fig. 2 shows the steps involved in transforming a geometric object first into a Shapes program, and then, at runtime, into triangles the GPU can render. The object is interactively created on a conventional 3D modeling package, and the sequence of operations used to create it is translated into a Shapes program. This is directly analogous to the way the 2D modeling program Adobe Illustrator converts user strokes and interactions into a PostScript program.

The compiled Shapes program is used to compute the exact representation of the CSG intersection curves. In our current implementation this can take anywhere from a few seconds up to 5 or 10 minutes so this process is done offline after the object is modeled.

Next the domain is statically decomposed into a small number of triangular domain regions called static domain triangles. These domain regions have the property that they can be easily subdivided and sampled at whatever resolution is necessary at run time.

The compiled Shapes code, the intersection curve descriptors, and the static domain triangles are bundled together to make a runtime Shapes object which contains everything necessary to triangulate the surface at run time.

The last phase of processing occurs at run time just before the object needs to be displayed. First, the implicit curves of intersection are used to dynamically trim the domain of the general parametric surface function into regions. The particular CSG operation used to create the object determines which of these regions will be triangulated and rendered. Then the domain regions to be rendered are subdivided into smaller, but still relatively large triangles, called domain sampling triangles. The computation of curve intersection points and the dynamic subdivision of the static domain triangles into domain sampling triangles is done on the CPU because it is cumbersome to implement these operations on current GPU's.

The domain sampling triangles are passed to the GPU, along with the compiled Shapes program representing the surface. The Shapes program is evaluated on the GPU at a large number of points in each sampling triangle and the resulting n-dimensional surface points become vertices in a triangle mesh. A typical vertex might be $[x,y,z,n_x,n_y,n_z,u,v]^T$ but one could have additional elements for color, reflection parameters, etc. Shapes places no limitations on the dimension or contents of this vector.

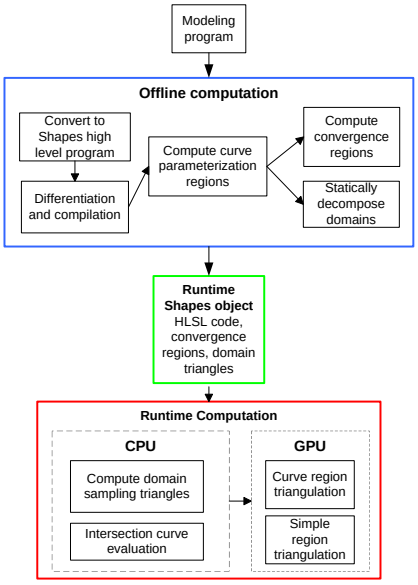

Figure 2: *Flow diagram of procedural model processing*

4. Finding Exact Curves of Intersection

In our system surfaces are parametric functions of two variables. In a typical CSG operation two surfaces defined by

$$f_1(u_0,u_1) = [f_{1x}(u_0,u_1), f_{1y}(u_0,u_1), f_{1z}(u_0,u_1)]^T$$
$$f_2(u_2,u_3) = [f_{2x}(u_2,u_3), f_{2y}(u_2,u_3), f_{2z}(u_2,u_3)]^T$$

are intersected. The functions $f_i : R^2 \rightarrow R^3$ are assumed to be non self-intersecting with a Lipschitz first derivative everywhere except perhaps at a set of points which can be found with minimal computation as for example would be the case if two curve segments were joined with a known first derivative discontinuity. This is essentially equivalent to requiring the functions to be piecewise C^2.

Each CSG operation gives rise to a set of closed 4D intersection curves in the variables u_0,u_1,u_2,u_3 defined by the implicit function $f : R^4 \rightarrow R^3$:

$$f(u_0,u_1,u_2,u_3) = f_1(u_0,u_1) - f_2(u_2,u_3) = [0,0,0]^T \quad (1)$$

The exact representation of the intersection curves is computed in two steps. First we use the implicit function theorem to partition the intersection curve into regions each of which can be parameterized by at least one variable in each domain. In a region parameterized by variable u_i the implicit function theorem guarantees that there exists a function $\mathbf{g}(u_i) : R^1 \rightarrow R^3$ which defines the remaining three variables in terms of u_i:

$$\mathbf{g}(u_i) = [g_{u_j}(u_i), g_{u_k}(u_i), g_{u_l}(u_i)]^T \quad j \neq k \neq l \neq i \quad (2)$$

More generally we will write

$$\mathbf{g}(u_{ind}) = \mathbf{u}_{dep} \quad (3)$$

where $\mathbf{u}_{dep}$ is a 3 vector of functions that define the dependent variables in terms of the scalar parameterizing variable u_{ind}. The implicit function theorem asserts that such a function exists but does not offer any suggestions about how it may be computed.

The second step of the algorithm finds an explicit, computable representation of this implicit function that is valid over the region parameterized by variable u_i. This step is the fundamentally new part of our algorithm and is described in section 4.1.

The parameterization regions are computed using an algorithm similar to that in [Sny92a]. The differences aren't very great, although our slightly stronger parameterization requirements simplify our real time triangulation algorithm. However this part of the work is not what we consider our most significant contribution. We include this detailed description here because it forms the foundation for the next section and because [Sny92a] is out of print and so may be difficult to find. For those readers familiar with the parameterization algorithm we advise skipping to section 4.1.

In Snyder's algorithm the 4D domain was subdivided into

a set of boxes which completely enclosed the intersection curve. Inside each box the curve could be parameterized by at least one variable. Snyder then computed intersections of the intersection curve with the box faces and approximated the true curve as a polyline.

In our algorithm we subdivide the 4D domain into boxes which completely enclose the intersection curve. Inside each box the curve can be parameterized by at least one variable in each domain, i.e., the curve must be parameterizable by at least two variables. This slightly stronger parameterizability requirement is necessary for the real time triangulation algorithm. We then fuse boxes of common parameterization into larger parameterization regions (Fig. 3). These parameterization regions are used in the next phase of our algorithm (section 4.1) where we find a computable representation of the implicit function.

f, the intersection curve function, is defined over a 4D interval box

$$\underline{\overline{\mathbf{u}}} = [\underline{\overline{u}}_0, \underline{\overline{u}}_1, \underline{\overline{u}}_2, \underline{\overline{u}}_3]^T \tag{4}$$

where $\underline{\overline{\mathbf{u}}}$ is initialized to the entire domain of the surface. The notation $\underline{\overline{x}}$ indicates an interval over the variable x where $\overline{x}$ is the upper and $\underline{x}$ the lower bound (see [HW04] for a good recent introduction to interval analysis or [Sny92b, Duf92] for applications to graphics). The initial vector interval $\underline{\overline{\mathbf{u}}}$ is then recursively subdivided until all boxes $\underline{\overline{\mathbf{p}}}_b$ are found that satisfy

$$0 \in f(\underline{\overline{\mathbf{p}}}_b) \tag{5}$$

as well as

$$\begin{array}{c}\{0 \notin \mathrm{Det}(\mathbf{D}_{-u_1} f(\underline{\overline{\mathbf{p}}}_b)) \quad or \quad 0 \notin \mathrm{Det}(\mathbf{D}_{-u_0} f(\underline{\overline{\mathbf{p}}}_b))\} \\ and \{0 \notin \mathrm{Det}(\mathbf{D}_{-u_2} f(\underline{\overline{\mathbf{p}}}_b)) \quad or \quad 0 \notin \mathrm{Det}(\mathbf{D}_{-u_3} f(\underline{\overline{\mathbf{p}}}_b))\}\end{array} \tag{6}$$

where the dependent derivative, $\mathbf{D}_{-u_i} f$, is $\mathbf{D} f$ minus the column containing derivatives with respect to variable u_i. For example, $\mathbf{D}_{-u_2} f$ is:

$$\mathbf{D}_{-u_2} f = \begin{bmatrix} \frac{\partial f_x}{\partial u_0} & \frac{\partial f_x}{\partial u_1} & \frac{\partial f_x}{\partial u_3} \\ \frac{\partial f_y}{\partial u_0} & \frac{\partial f_y}{\partial u_1} & \frac{\partial f_y}{\partial u_3} \\ \frac{\partial f_z}{\partial u_0} & \frac{\partial f_z}{\partial u_1} & \frac{\partial f_z}{\partial u_3} \end{bmatrix} \tag{7}$$

The conditions of (6) guarantee parameterizability. By the implicit function theorem if $0 \notin \mathbf{D}_{-u_i}(f(\underline{\overline{\mathbf{p}}}_b))$ then there exists a function $\mathbf{g}(u_i) = \mathbf{u}_{dep}$, i.e., we can parameterize the curve by variable u_i everywhere in $\underline{\overline{\mathbf{p}}}_b$. Since we require that at least one of the $\mathbf{D}_{-u_i}$ be non-singular in each domain we are guaranteed that we can parameterize every part of the curve by at least one of the domain variables.

In general, the conditions of the implicit function theorem will not be satisfied at points where the two surfaces are tangent or at points where the parameterization itself is singular as, for example, occurs at the north and south poles of the simple parametric definition of a sphere. Tangent surfaces give rise to a host of numerical robustness issues and one can model many interesting surfaces without allowing them. As a consequence CSG operations on tangent surfaces will not be considered further in this paper, although extending our work to handle this case is an interesting research problem. Similarly our system does not currently allow CSG operations which result in implicit curves of intersection passing through singular parameterization points.

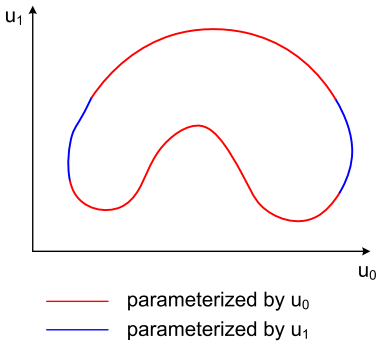

Figure 3: *An intersection curve in one domain is partitioned into regions parameterizable by at least one of the variables.*

If more than two objects are involved in a CSG operation then it is possible that curves of intersection might themselves intersect. Our current implementation does not handle this case. In the absence of tangencies and singular parameterizations this problem is no more difficult to solve than that of computing the intersection of two $R^2 \rightarrow R^3$ functions but adding this extra level of functionality significantly complicates the code. This is more of an engineering than a theoretical limitation which we plan to address in the next version of our system.

Condition (5) eliminates those boxes which cannot contain the intersection curve: if $f \neq 0$ everywhere in the box then clearly the curve cannot be in the box so it is discarded and not subdivided further. Because the range bounds on f are not tight it is possible that the box does not contain the curve even though condition (5) is met. We eliminate these false boxes by computing the intersection of the curve with each box face using interval Newton root finding. If there are no face intersections then the box does not contain the curve and it is discarded. A few additional steps are performed to minimize box size but they are not relevant to our discussion. The details can be found in [Sny92a].

The boxes which have satisfied (5), (6), and the face intersection test are linked together into connected components by matching face intersection points. The result is a linked list of 4D boxes which completely contain the curve. Sequential boxes of common parameterization are fused to form a single parameterization region which is defined by an interval in the parameterizing variable $\underline{\overline{u}}_i$ (Fig. 3).

4.1. Proving Convergence of Newton Iteration

While techniques for computing parameterizability have been known for some time, up to now there has not been an efficient way to use this information to evaluate points on the curve rapidly enough to be useful for real time rendering. Our new algorithm uses Newton iteration to solve the implicit curve equation at run time. Convergence of the Newton iteration is assured by finding regions of guaranteed convergence over intervals of the parameterizing variable, and storing these regions as part of the Shapes runtime object.

Assume we have a parameterization region $\underline{u}_i$ and that we wish to solve for a point on the curve corresponding to the parametric value $c \in \underline{u}_i$. The parametric function $\mathbf{g}(\underline{u}_i)$ gives the unique 3 vector $\mathbf{u}_{dep}$ that satisfies

$$f(\mathbf{c}) = 0 \tag{8}$$

where the notation

$$\mathbf{c} = [c, \mathbf{u}_{dep}]$$

denotes a 4 vector consisting of the independent scalar parameterizing variable, in this case equal to c, and the 3 vector of dependent variables $\mathbf{u}_{dep}$. For example, if the parameterizing variable is u_1 and $\mathbf{u}_{dep} = [4,5,6]$ then

$$\mathbf{c} = [c, \mathbf{u}_{dep}] = [4, c, 5, 6]$$

The parametric function $\mathbf{g}(\underline{u}_i)$ is constructed by partitioning $\underline{u}_i$ into intervals of guaranteed convergence $\underline{u}_{con_k}$ each of which has an associated dependent variable starting point $\mathbf{u}_{dep_k}$. To compute a curve point $[c, \mathbf{u}_{dep}]$ the appropriate convergence region

$$\{\underline{u}_{con_k} | c \in \underline{u}_{con_k}\} \tag{9}$$

is found. Then $[c, \mathbf{u}_{dep_k}]$ is used as the starting point for the Newton iteration

$$\mathbf{u}_{dep_{j+1}} = \mathbf{u}_{dep_j} - \mathbf{h}_j \tag{10}$$

where $\mathbf{h}_j$ is

$$\mathbf{h}_j = \left[\mathbf{D}_{-u_i} f([c, \mathbf{u}_{dep_j}])\right]^{-1} f([c, \mathbf{u}_{dep_j}])$$

On average three to four iterations are enough to get 7-8 digits of accuracy. This iteration is evaluated at run time to compute points on the intersection curve.

To compute the regions of guaranteed convergence, $\underline{u}_{con_k}$, one can use an interval extension of Kantorovich's theorem (see appendix B), which gives sufficient conditions to guarantee convergence of Newton iteration. The interval extension is essentially equivalent to Hubbard's balls of convergence [HH02] used in his constructive proof of the implicit function theorem. The result of the interval extension is a function $\overline{K}$

$$\overline{\mathbf{u}} = [\underline{u}_i, \underline{\mathbf{u}}_{dep}]$$
$$\overline{K}(\overline{\mathbf{u}}) = \|f(\overline{\mathbf{u}})\| \left\| \left[\mathbf{D}_{-u_i} f(\overline{\mathbf{u}})\right]^{-1} \right\|^2 \overline{M} \tag{11}$$

which depends on f, its derivative, and the Lipschitz first derivative bound $\overline{M}$. If $\overline{K}(\overline{\mathbf{u}}) < .5$ then for any $u_{ind} \in \underline{\mathbf{u}}_i, u_{dep0} \in \underline{\mathbf{u}}_{dep}$ Newton iteration starting from u_{dep0} will converge quadratically to the solution of $f([u_{ind}, \mathbf{u}_{dep}]) = 0$.

One could attempt to use the interval extension of Kantorovich's theorem directly to find regions of guaranteed convergence. Unfortunately Kantorovich's theorem provides only sufficient conditions for convergence and in general yields quite pessimistic estimates of the size of convergence regions. For the surfaces we have modeled in our system thousands of these convergence regions would be required to completely cover a single curve of intersection. The convergence regions are required to compute curve points at run time so minimizing their number reduces the memory footprint of the procedural object. Thousands of regions would take up far too much space to be practical.

While a naive application of Kantorovich's theorem is not practical, it does form the basis for our new algorithm which computes regions of convergence that are orders of magnitude larger than those predicted by the interval form of Kantorovich's theorem.

4.2. Finding Large Regions of Convergence

Assume that we want to prove convergence from starting interval

$$\overline{\mathbf{u}}_0 = [\underline{u}_i, \mathbf{u}_{dep_0}] \tag{12}$$

but that $\overline{K}([\underline{u}_i, \mathbf{u}_{dep_0}]) > .5$. Compute a new interval box $\overline{\mathbf{u}}_1$ which is the image of $\overline{\mathbf{u}}_0$ under the Newton transformation

$$\begin{aligned} \overline{\mathbf{h}}_0 &= \left[\mathbf{D}_{-ui} f([c, \underline{\overline{\mathbf{u}}}_{dep_0}])\right]^{-1} f([c, \underline{\overline{\mathbf{u}}}_{dep_0}]) \\ \underline{\overline{\mathbf{u}}}_{dep_1} &= \mathbf{u}_{dep_0} + \underline{\overline{\mathbf{h}}}_0 \\ \overline{\mathbf{u}}_1 &= [\underline{u}_i, \underline{\overline{\mathbf{u}}}_{dep_1}] \end{aligned} \tag{13}$$

If $\overline{K}(\overline{\mathbf{u}}_1) \geq .5$ continue computing points $\overline{\mathbf{u}}_j$ until at some step k, $\overline{K}(\overline{\mathbf{u}}_k) < .5$. By the interval extension of Kantorovich's theorem we know that Newton iteration starting from any point in $\overline{\mathbf{u}}_k$ will converge. Since all of the points in $\overline{\mathbf{u}}_{k-1}$ map into $\overline{\mathbf{u}}_k$ then all points in $\overline{\mathbf{u}}_{k-1}$ must converge. By continuing this argument backwards through the $\overline{\mathbf{u}}_i$ we arrive at the conclusion that every point in $\overline{\mathbf{u}}_0$ will converge.

If after some maximum number of steps, m, none of the $\overline{\mathbf{u}}_j$ satisfy the condition

$$\overline{K}(\overline{\mathbf{u}}_k) < .5 \quad k = 0..m \tag{14}$$

then $\underline{u}_i$ is split in half and new starting points are computed at the midpoint of each half using standard interval Newton

root finding. The test is then performed recursively on each half:

```
findConvergenceRegions(interval up, list cR){
   us = curveDependentValues(up.mid)
   if(convergence(up,us))
      append (up(interval),us) to cR
   else
      lowHalf = (up.low,up.mid)
      highHalf = (up.mid,up.high)
      //attempt to prove conver-
gence on new intervals
      //with new starting points
      findConvergenceRegions (lowHalf, cR)
      findConvergenceRegions (highHalf, cR)
}
boolean convergence(interval u_p, vec-
tor u_s){
   up1 = u_p
   while(K(u_p1,u_s)<alpha*K(u_p,u_s)){
      hi = inverse(D_up(f(u_p,u_s))f(u_p,u_s)
      u_p1 = u_p - hi
      if(K(u_p1) < .5 && in-
ParamBox(u_p1)) return true
   }
   low = (u_p.low,u_p.mid)
   high = (u_p.mid,u_p.high)
   //attempt to prove conver-
gence on new intervals
   //using the same starting point
   return convergence(low,u_s)
          && convergence(high,u_s)
}
```

The Lipschitz first derivative and parameterizability conditions (6) guarantee that eventually we will find convergence regions of non-zero size. Since $\|\mathbf{D}^{-1}f\|$ and M are both bounded in the parameterizability boxes which completely enclose the curve but $\|f([\underline{u}_i,\mathbf{u}_{dep_k}])\|$ becomes arbitrarily small as we decrease the width of $\underline{u}_i$ then

$$\overline{K}([\underline{u}_i,\mathbf{u}_{dep_k}]) \rightarrow 0 \text{ as } \text{width}(\underline{u}_i) \rightarrow 0 \qquad (15)$$

Uniqueness of the point of convergence is guaranteed by the parameterizability condition associated with each parametric region. Assume the convergence region $\underline{u}_0$ is part of parametric region $\underline{u}_p$ and that $\underline{u}_k$ is the image of $\underline{u}_0$ after k Newton steps. For $\text{width}(\underline{u}_i)$ small enough there will be such a $\underline{u}_k$. If

$$\overline{K}(\underline{\mathbf{u}}_k) < .5 \quad \text{and} \quad \underline{\mathbf{u}}_k \subseteq \underline{\mathbf{p}}_b \qquad (16)$$

where $\underline{\mathbf{p}}_b$ is a parameterizability box containing some part of the curve parameterized by $\underline{u}_p$ then for every $u_c \in \underline{u}_0$ Newton iteration converges to some $\mathbf{u}_{dep} \in \underline{\mathbf{p}}_b$. Since the curve is parameterizable everywhere in $\underline{\mathbf{p}}_b$ then $\mathbf{u}_{dep}$ is the unique point such that

$$f([u_c,\mathbf{u}_{dep}]) = 0 \qquad (17)$$

Overestimation of bounds due to interval analysis combined with the widening caused by outward rounding increase the size of the $\underline{\mathbf{u}}_j$ at each iteration which tends to increase $\overline{K}(\underline{\mathbf{u}}_j)$ when the boxes become very wide. But the $\underline{\mathbf{u}}_j$ are also typically getting closer to the solution curve at each iteration which reduces $\|f(\underline{\mathbf{u}}_j)\|$ and frequently $\overline{K}(\underline{\mathbf{u}}_j)$ as well. It is this effect which makes `convergence()`so effective at proving convergence over large regions.

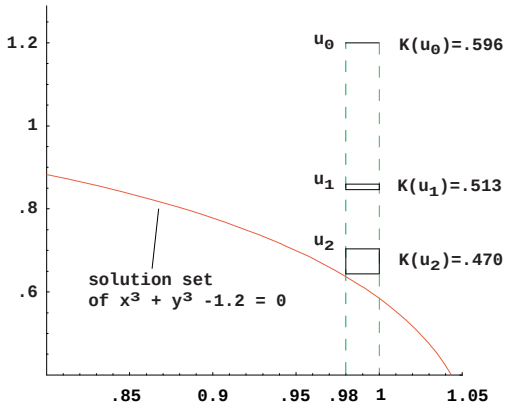

Figure 4: *Proving convergence of Newton iteration for every point in* $[\{.98,1\},1.2]$ *to the solution of* $x^3+y^3-1.2=0$. *x is the parameterizing variable and y is the dependent variable.* $\overline{K}(\underline{\mathbf{u}}_0) > .5$ *so Kantorovich's theorem isn't satisfied. In the next step* $\underline{\mathbf{u}}_0$ *is transformed by a Newton step into* $\underline{\mathbf{u}}_1$. $\overline{K}(\underline{\mathbf{u}}_1) < \overline{K}(\underline{\mathbf{u}}_0)$ *even though the dependent variable interval of* $\underline{\mathbf{u}}_1$ *is wider than that of* $\underline{\mathbf{u}}_0$. *Finally,* $\overline{K}(\underline{\mathbf{u}}_2) < .5$ *so convergence from every point in* $\underline{\mathbf{u}}_2$,$\underline{\mathbf{u}}_1$, *and* $\underline{\mathbf{u}}_0$ *is guaranteed.*

A simplified 2D example illustrates both phenomena described above (Fig. 4). Function `convergence()`is applied to the function $x^3+y^3-1.2=0$ parameterized by x in the range $\{.98,1\}$ with dependent variable starting point $y=1.2$. Initially, at $\underline{\mathbf{u}}_0=[\{.98,1\},1.2)]$, $\overline{K}(\underline{\mathbf{u}}_0)$ is .596, because $\underline{\mathbf{u}}_0$ is far away from the solution curve. Each succeeding $\underline{\mathbf{u}}_j$ comes closer to the curve but the width of the dependent variable interval becomes much larger as well: $\underline{\mathbf{u}}_1 = [\{.98,1\},\{.846,.86\}]$,$\underline{\mathbf{u}}_2 = [\{.98,1\},\{.643,.703\}]$. The reduction in $\|f(\underline{\mathbf{u}}_j)\|$ dominates for the first few iterations so that by the second iteration $\overline{K}(\underline{\mathbf{u}}_2) = .470 < .5$ and convergence from $\underline{\mathbf{u}}_0$ is proven.

5. Triangulation

There are two different types of computation performed at runtime: evaluating 4D intersection curve points using (10) and evaluating surface points and normals. Curve points are evaluated on the CPU because it is difficult to make this computation run efficiently on the GPU. All surface points and normals are computed on the GPU.

For the objects we have constructed to date we can evalu-

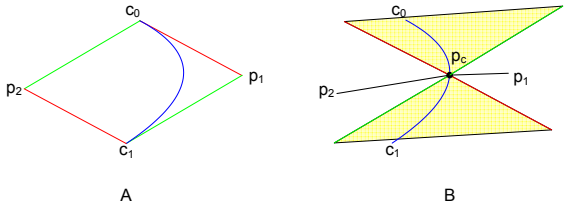

Figure 5: *A) Intersections of the min and max values of the 2D tangent over the curve segment gives visibility points p_1 and p_2. B) Every point on the curve must lie in the shaded cone bounded by the tangent lines. All points of the curve segment are visible from p_1 and p_2.*

ate approximately 200,000 4D curve points per second. For some applications this is fast enough to get real time update rates. With many objects visible on the screen this rate is not quite fast enough. In this case the curve can be sampled densely and the curve points cached. This is done just once when an object first changes from a dormant state to an active state, meaning from an invisible to a visible or potentially visible soon state. For all succeeding frames curve points are read from the cache rather than computed. Most objects will require only a few hundred curve points to render intersection curves accurately so activation time will be a few milliseconds. When the object goes back to a dormant state the cache space is reclaimed and used for another object.

CSG operations are effected by triangulating only that part of the domain which corresponds to visible parts of the CSG surface. The first phase of triangulation is performed offline. The parts of the domain to be triangulated are subdivided into a small number of triangular domain regions of two types. Curve visibility triangles, v_i, are bounded along one side by an intersection curve and along the other two edges by line segments. Simple triangles, s_i are bounded on all three sides by line segments.

The second phase of triangulation is performed at runtime just before the object is to be rendered. The static domain triangles are subdivided into smaller domain sampling triangles if necessary and these triangles are passed to the GPU for evaluation. The GPU uses the compiled HLSL Shapes code representing the surface and geometry instancing, a feature available in cards that support the DirectX Vertex Shader 3.0 model, to evaluate the surface. After the surface is rendered the domain sampling triangles are discarded.

5.1. Offline Processing

Curve visibility triangles, v_i, are computed for each parametric segment of every interesection curve. There are two visibility triangles for each segment (Fig. 5). Each triangle shares two 2D vertices, c_0, c_1 on the curve. Two additional 2D points p_1,p_2 not on the curve are the apices of the two triangles. The part of the curve between c_0 and c_1 is defined by an interval over a single parameterizing variable $\underline{u}_p$. Each apex p_i has the property that every point of the curve segment associated with the visibility triangle is visible from either p_i.

Visibility regions are easily computed by using implicit differentiation (see Appendix C) over the interval $\underline{u}_p$ to compute the extreme values of the 2D tangent. The intersection of the min and max tangent lines gives the two visibility points p_1,p_2. By the mean value theorem if $u_c \in \underline{u}_p$ the curve must lie entirely within the shaded region bounded by the max and min slope lines centered at p_c. A line segment from p_i to p_c clearly will lie outside this region except exactly at the endpoints of $\underline{u}_p$ so there will be a single intersection with the curve for any $u_c \in \underline{u}_p$.

Edges are added between the endpoints of each parametric segment and the domain is triangulated with constrained Delaunay triangulation. Visibility triangles are then subdivided until they do not intersect any other visibility triangle edges.

5.2. Runtime Processing

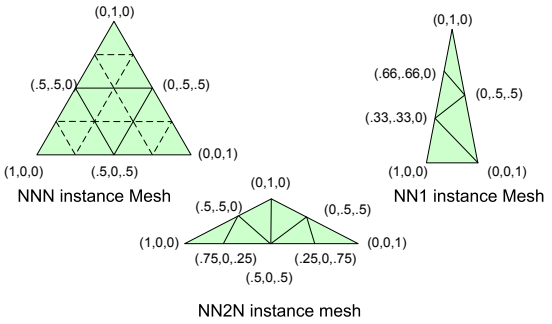

Figure 6: *Barycentric coordinates stored in the instance mesh*

Runtime triangulation uses vertex shaders and geometry instancing to move most of the work of surface evaluation onto the GPU. Geometry instancing is a kind of primitive looping construct that has as input two vertex streams. The instance vertex stream contains vertices of a triangle mesh called the instance mesh. The per-instance vertex stream has one vertex per instance to be displayed. Conceptually the looping works like this:

```
foreach(vertex v in the per-instance ver-
tex stream)
    foreach(vertex iv in the instance ver-
tex stream)
        run the vertex shader with in-
puts v,iv
```

Instance mesh vertices contain the barycentric coordinates of sample points defined on a canonical base triangle. There are three different types of instance meshes (low sampling level versions of these are shown in Fig. 6). The NNN type is used for all non curve triangles, has N vertices along each edge and roughly $.5N^2$ interior vertices. The $NN1$ type is used for curve visibility triangles bounded along one side by an intersection curve. It has N vertices along two edges and 2 vertices along the curve edge. The $NN2N$ type is for triangles generated during the subdivision of curve visibility triangles. This type lies inside the curve visibility triangle but are not bounded on any side by an intersection curve. It has roughly N vertices along the two short edges and $2N$ vertices along the long edge.

At each frame time we perform the following steps on the CPU. 4D points along the CSG intersection curve are computed using (10), or cached points are read from the cache. These points are used to subdivide the curve visibility triangles. Because the intersection curve points are shared between the intersecting surfaces the triangles along the curve boundary will be completely consistent. (Fig. 7). The subdivided curve triangles and all non-curve static domain triangles are dynamically subdivided into domain sampling triangles. (Fig. 8) Edge length is computed in the parametric space of the domain and subdivision continues until it is below some user defined minimum value. The three 2D vertices of each resulting sampling triangle make up one vertex in the per-instance stream. The sampling triangle vertices are copied into the per-instance vertex stream and this, along with the instance meshes, is sent to the GPU for evaluation of surface points. After each object is rendered the per-instance vertex buffer used to store the domain sampling triangle data is discarded.

This snippet of HLSL code shows how a point in the domain of the surface function is computed on the GPU from the per-instance vertex data (variables p0,p1,p2) and the instance mesh barycentric coordinates (variable bary):

```
struct VS_IN{
     float3 bary,
     float2 p0,
     float2 p1,
     float2 p2};
main( VS_IN In ){
  float2 pos = In.bary.x*In.p0
             + In.bary.y*In.p1
             + In.bary.z*In.p2;
}
```

This 2D domain point is then used by the HLSL surface function to evaluate the surface position and normal.

For any one object all 3 instance meshes of types NNN, $NN1$, $NN2N$ which are used to render the object have the same value of N, the number of sample points along an edge. We do this for efficiency and simplicity. If N could vary on a per sampling triangle basis then, in order to ensure consistent sampling along shared edges, we would have to create

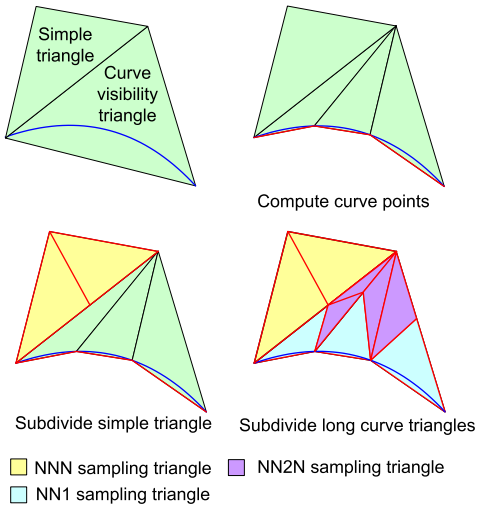

Figure 7: *Runtime subdivision of static domain curve and simple triangles. Red edges are not subdivided further on the CPU. Edges are subdivided based on parametric length. The subdivision length for curve edges is 1/N the length for all other edges because 2D domain coordinates of all curve samples are computed on the CPU. Curve edges are proportionally much shorter than they appear here. The scale has been changed to make the curve triangle subdivision process easier to see.*

instance meshes with all possible combinations of edge sampling numbers. Even worse we would have to set up the vertex streams and issue a DirectX drawPrimitive call for each different resolution instance mesh used to render the object. Both of these operations have high overhead and their use should be minimized in order to achieve high frame rates.

6. Results

All of the timings in the results section were measured on a Pentium Xeon 2.6 GHz processor with an NVidia 6800 Ultra graphics card. The two most important metrics of the new procedural object representation are size and rendering speed. Objects exist in two states: dormant and active. A dormant object contains only static object data and is the smallest possible representation of the object. An active object has both the static data plus cached intersection curve points, if any. As mentioned in section 5 our current implementation computes approximately 200,000 intersection curve points per second. This rate is high enough for interactive rendering of a few objects but when the number of objects increases it is better to cache the 4D intersection curve points at the instant the object first becomes active, i.e., visibile or potentially visible soon and then to use the cached points when rendering. When the object becomes dormant

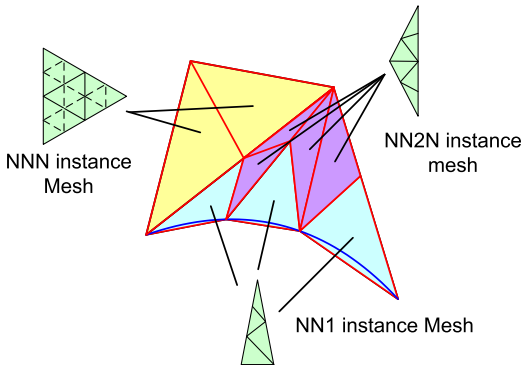

Figure 8: *Instance meshes which will be evaluated on the GPU for each type of sampling triangle. Simple domain sampling triangles use an NNN instance mesh. Domain sampling triangles bounded by a curve edge use an NN1 instance mesh. Thin triangles not bounded by a curve edge are sometimes generated in the curve triangle subdivision, in order to ensure consistency of sampling along shared edges. They use an NN2N instance mesh.*

again the space for the point cache is reclaimed. This is the method we have used to measure all our rendering speeds.

All of the objects shown in the illustrations render at approximately 20 million triangles/second. Each triangle is texture mapped and environment mapped with a DirectX 9.0 pixel shader that is approximately 40 lines of HLSL code.

While it is conceivable for a system to have tens of thousands of objects in a dormant state only a few hundred, perhaps as many as a thousand, will be active at any time. The memory required to cache the intersection curve points of even a thousand active objects is less than 10 MBytes. This is a small fraction of the memory on modern PC's or next generation game consoles so it will not be the limiting factor in memory usage. In contrast, the dormant objects could take up a considerable amount of memory so we will concentrate on computing their size.

Each object is made up of a number of surfaces and associated CSG operations. The dormant size of an object is determined by the number of lines of code required to represent each surface function and its derivatives, and the number of convergence regions and static domain triangles per surface.

There are two types of code: the HLSL surface and normal function, and the C# derivative function. Both of these pieces of code share whatever 2D spline control points were used to make the extrusions or surfaces of revolution so we count this only once. The derivative code is shared between the two surfaces involved in a CSG operation so we will also count

name	HLSL	C#	data	total	t_{pr}	t_{conv}
speaker	720	2K	6.2K	8.9K	3s	76s
wheel 0	432	960	6.9K	8.2K	57s	104s
wheel 1	432	960	10K	11.4K	63s	148s
wheel 2	432	960	5.6K	7K	121s	270s
tire	288	960	10.7K	11.9K	8s	683s

Table 1: *Computed memory size, in bytes, of procedural objects and offline processing time, in seconds, to compute parameterization regions, t_{pr}, and to prove convergence, t_{conv}. The data column includes space required for the convergence regions, the static domain triangles, and the 2D splines used in surfaces of revolution or extrusion.*

it only once. The code for all the surfaces is nearly the same size: roughly 36 lines of HLSL code and 80 lines of C# code.

Both the HLSL and C# code consist almost entirely of statements of the form $a = b$ op c so there should be a nearly one to one mapping from lines of code to number of assembly instructions. Since HLSL compiles to proprietary object code in the graphics card driver and C# compiles to an intermediate language which is jit'ed at load time, we can't measure the true size of the object code. We will assume each assembly instruction is 6 bytes long on average and that each source line translates to a single assembly instruction.

Table 1 shows the computed storage requirements and offline processing time for each of the objects in the illustrations. To put these numbers in some perspective let's compare them to the memory required to store a polygon mesh. If we assume each vertex in a polygon mesh has components $x, y, z, n_x, n_y, n_z, u, v$ then each vertex requires 32 bytes. Assuming there are roughly twice as many triangles as vertices and that the triangle connectivity information takes 6 bytes (three 2 byte indices) each triangle in the mesh requires roughly 22 bytes. The memory required for wheel 2 in Fig.13, 7KBytes, is equivalent to that required for 320 triangles. Clearly this wheel could not be adequately represented by 320 triangles.

The system does a surprisingly good job of minimizing the number of parametrization regions required for each closed intersection curve. Fig. (9) shows a typical domain triangulation and parameterization. Both intersection curves have 4 parameterization regions, which is the minimum theoretically possible in this case.

Figures 11 and 10 show histograms of the number of parameterization regions per intersection curve and the number of convergence regions per parameterization region, respectively. These histograms represent the combined statistics of

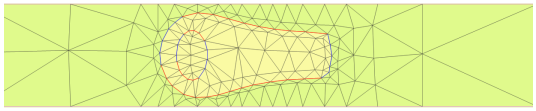

Figure 9: *Domain parameterization regions and triangulation for a typical procedural model.*

all the objects shown in the illustrations. Most intersection curves have 5 parameterization regions or less. Most parameterization regions have only a single convergence region, which is the best possible result, with a relatively small number having more.

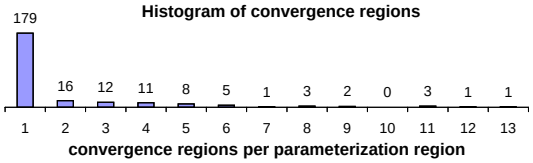

Figure 10: *Histogram of the number of convergence regions per parameterization region. Ideally all parameterization regions would have just one convergence region. For our dataset this is true in the vast majority of cases.*

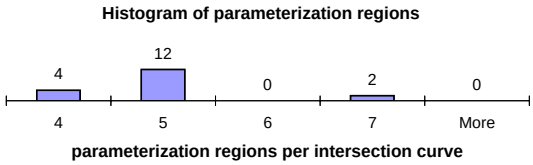

Figure 11: *Histogram of the number of parameterization regions per intersection curve. For closed curves that do not cross domain boundaries at least 4 regions will be required. Most intersection curves in our dataset have just slightly more than 4 parameterization regions.*

7. Conclusion

Our new representation for generative CSG models is very compact, resolution independent, and renders quickly on modern GPU's. The objects in the illustrations have a memory footprint of only 7-11 KBytes and render at approximately 20 million triangles/second. These highly compact objects are ideally suited for memory constrained architectures, such as game consoles, or bandwidth constrained applications, such as online games.

Future graphics architectures should be even better suited to render Shapes programs. With relatively minor changes to the architecture of current GPU's it should be possible to run the intersection curve evaluation on the GPU which would make it much faster. DirectX10 style geometry shaders should make run time triangulation more flexible resulting in triangulations which take fewer triangles and which render faster.

We have only used the exact piecewise parametric representation for implicit curves of intersection between parametric surfaces but the method is, in principle, quite general. For example, it should be possible to find piecewise parametric forms of implicit surfaces that satisfy the conditions laid out in section 4. It should also be possible to find a piecewise parametric form of the implicit curve of intersection between two implicit surfaces. However, since the parameterizability and convergence proof computations grow rapidly as the dimensionality of the problem increases more research will be needed to see how well this algorithm will scale.

The proof of large regions of convergence for Newton iteration, which applies to a very broad class of functions, is also completely general and should have applications outside of real time graphics rendering.

References

[AD03] ADAMS B., DUTRE P.: Interactive boolean operations on surfel-bounded solids. *SIGGRAPH '03: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2003), 651–656. 2

[BKZ01] BIERMANN H., KRISTJANSSON D., ZORIN D.: Approximate boolean operations on free-form solids. *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), 185–194. 2

[CC78] CATMULL E., CLARK J.: Recursively generated B-spline surfaces on arbitrary topological meshes. 350–355. 2

[DKT98] DEROSE T., KASS M., TRUONG T.: Subdivision surfaces in character animation. *Proceedings of SIGGRAPH 98* (1998), 85–94. 2

[DS78] DOO D., SABIN M.: Behaviour of recursive division surfaces near extraordinary points. 356–360. 2

[Duf92] DUFF T.: Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics 26*, 2 (1992), 131–139. 4

[HH02] HUBBARD J. H., HUBBARD B. B.: *Vector Calculus, Linear Algebra, and Differential Forms A Unified Approach*. Prentice Hall, 2002. 5, 11

[HW04] HANSEN E., WALSTER G. W.: *Global Optimization Using Interval Analysis*. Marcel Dekker, 2004. 4

[Owe05] OWENS J.: Streaming architectures and technology trends. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, 2005, ch. 29. 1

[PKKG03] PAULY M., KAISER R., KOBBELT L., GROSS M.: Shape modeling with point sampled geometry. *SIGGRAPH '03: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2003), 641–650. 2

[SLJ03] STEWART N., LEACH G., JOHN S.: Improved csg rendering using overlap graph subtraction sequences. *International*

Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia - GRAPHITE 2003 (Feb 2003), 47–53. 2

[Sny92a] SNYDER J.: *Generative Modeling for Computer Graphics and CAD*. Academic Press, 1992. 1, 3, 4

[Sny92b] SNYDER J. M.: Interval analysis for computer graphics. *ACM Computer Graphics 26*, 2 (1992), 121–130. 4

[Sta98] STAM J.: Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. *Proceedings of SIGGRAPH 98* (1998), 395–404. 2

Appendix A: Notation

$\mathbf{D}f$	derivative of f with respect to all variables in the vector $\mathbf{u}$
$\mathbf{D}_{u_i}f$	derivative of f with respect to the single variable u_i
$\mathbf{D}_{-u_i}f$	$\mathbf{D}f$ with the column $\mathbf{D}_{u_i}f$ removed
$\overline{\underline{u}}$	interval in u with upper bound $\overline{u}$ and lower bound $\underline{u}$

Appendix B: Interval Extension of Kantorovich's Theorem

For the point version of Kantorovich's theorem see [HH02]. The interval extension is a relatively straighforward matter of substituting intervals for points and suitably extending the intervals over which the Lipschitz constant $\overline{M}$ must exist and the function f must have non-zero determinant.

Given a function $f([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}])$ assumed to be continously differentiable over a region as large as is necessary, a parameterizing variable $\overline{\underline{u}}_p$ which is the i_{th} argument of f, and dependent variable starting point $\overline{\underline{\mathbf{u}}}_{dep_0}$ compute the Newton displacement, $\overline{\underline{\mathbf{h}}}_0$,

$$\overline{\underline{\mathbf{h}}}_0 = \left[\mathbf{D}_{-ui} f([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}])\right]^{-1} f([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}]) \tag{18}$$

and the image, $\overline{\underline{\mathbf{u}}}_{dep_1}$, of $\overline{\underline{\mathbf{u}}}_{dep_0}$ after a single Newton step

$$\overline{\underline{\mathbf{u}}}_{dep_1} = \overline{\underline{\mathbf{u}}}_{dep_0} + \overline{\underline{\mathbf{h}}}_0 \tag{19}$$

Form the interval box

$$\overline{\underline{\mathbf{P}}}_0 = [\overline{\underline{u}}_p, [\overline{\underline{\mathbf{u}}}_{dep_1} + [\overline{\underline{r}}, \overline{\underline{r}}, \overline{\underline{r}}]]] \tag{20}$$

where $\overline{\underline{r}}$ is

$$\overline{\underline{r}} = \{-|\overline{\underline{\mathbf{h}}}_0|, |\overline{\underline{\mathbf{h}}}_0|\} \tag{21}$$

If $0 \notin \mathbf{D}_{-u_i}(f(\overline{\underline{\mathbf{P}}}_0))$ and if the Lipschitz constant $\overline{M}$

$$\|\mathbf{D}_{-u_i}(\mathbf{x}) - \mathbf{D}_{-u_i}(\mathbf{y})\| \leq \overline{M}\|\mathbf{x} - \mathbf{y}\| \tag{22}$$

exists for any $\mathbf{x}, \mathbf{y} \in \overline{\underline{\mathbf{P}}}_0$ then if

$$\overline{K}([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}]) = \|f([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}])\|\|\left[\mathbf{D}_{-u_i} f([\overline{\underline{u}}_p, \overline{\underline{\mathbf{u}}}_{dep_0}])\right]^{-1}\|^2 \overline{M} < .5 \tag{23}$$

then for any $u_p \in \overline{\underline{u}}_p, \mathbf{u}_{dep0} \in \overline{\underline{\mathbf{u}}}_{dep_0}$ Newton iteration starting from $\mathbf{u}_{dep0}$ will converge quadratically to the solution of $f([u_p, \mathbf{u}_{dep}]) = 0$.

Appendix C: Implicit Differentiation

See [HH02] for a derivation which applies to functions of any codimension. This derivation is only valid for functions with codimension 1, since this is the class of functions the CSG intersection curves belong to.

A point $\mathbf{u}$ on the curve can be written as $[u_i, g(u_i)]$ where

$$\mathbf{g}(u_i) = [g_{u_j}(u_i), g_{u_k}(u_i), g_{u_l}(u_i)]^T \quad j \neq k \neq l \neq i \tag{24}$$

is the parameterizing function for this part of the curve. For any $\mathbf{u}$ on the curve $f(\mathbf{u}) = 0$ and consequently $\mathbf{D}_{u_i} f(\mathbf{u}) = 0$ as well. By the chain rule

$$\mathbf{D}_{u_i} f([u_i, g(u_i)]) = \mathbf{D}f(\mathbf{u})\mathbf{D}_{u_i}[u_i, g(u_i)] = 0 \tag{25}$$

where

$$\mathbf{D}_{u_i}[u_i, g(u_i)] = \left[\frac{\partial u_0}{\partial u_i}, \frac{\partial u_1}{\partial u_i}, \frac{\partial u_2}{\partial u_i}, \frac{\partial u_3}{\partial u_i}\right]^T \tag{26}$$

The ith element in $\mathbf{D}_{u_i}[u_i, g(u_i)]$ will be 1. Reorder the variables so that u_i is the last element in $\mathbf{u}$. Then

$$\mathbf{D}_{u_i}[u_i, g(u_i)] = [\mathbf{D}_{u_i} g(u_i), 1]^T \tag{27}$$

and the columns of $\mathbf{D}f(\mathbf{u})$ will also be reordered so that $\mathbf{D}_{u_i} f(\mathbf{u})$ is the last column

$$\mathbf{D}f(\mathbf{u}) = \left[\mathbf{D}_{u_j} f(\mathbf{u}) \mid \mathbf{D}_{u_k} f(\mathbf{u}) \mid \mathbf{D}_{u_l} f(\mathbf{u}) \mid \mathbf{D}_{u_i} f(\mathbf{u})\right] \tag{28}$$

The first three columns of (28) are just $\mathbf{D}_{-u_i} f(\mathbf{u})$ so (25) becomes

$$[\mathbf{D}_{-u_i} f(\mathbf{u}) \mid \mathbf{D}_{u_i} f(\mathbf{u})]\,[\mathbf{D}_{u_i}[g(u_i)], 1]^T = 0 \tag{29}$$

$$\mathbf{D}_{-u_i} f(\mathbf{u})\mathbf{D}_{u_i}[g(u_i)] + \mathbf{D}_{u_i} f(\mathbf{u}) = 0 \tag{30}$$

$$\mathbf{D}_{u_i}[g(u_i)] = -\mathbf{D}_{-u_i}^{-1} f \mathbf{D}_{u_i} f(\mathbf{u}) \tag{31}$$

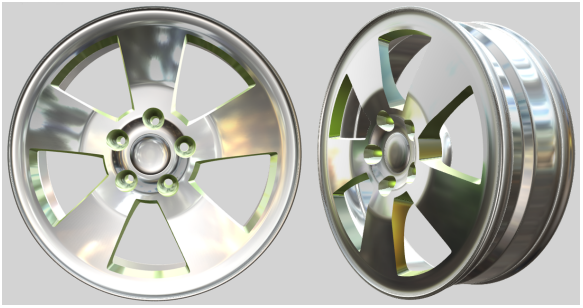

Figure 12: *Wheel 0: 8.2KBytes.*

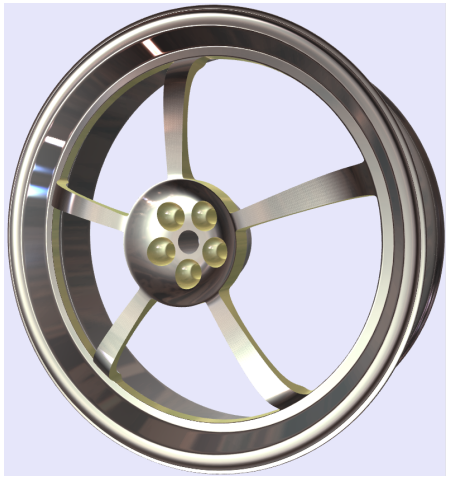

Figure 13: *Wheel 2: 7KBytes.*

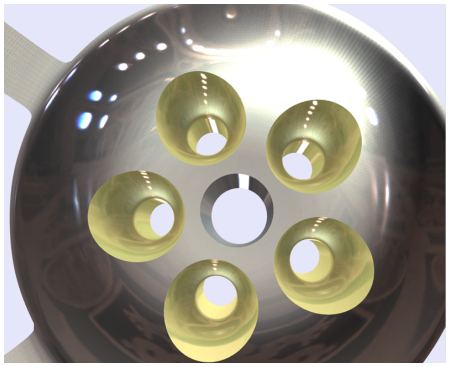

Figure 14: *Close up of bolt cutouts in Fig. 13.*

Figure 15: *Rear view of the wheel in Fig. 13*

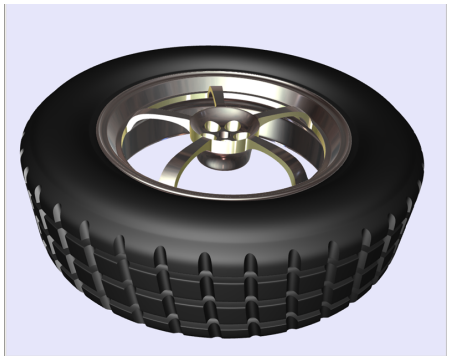

Figure 16: *The tire treads are geometric detail resulting from CSG operations, not a bump or normal map. Tire by itself: 11.9KBytes.*

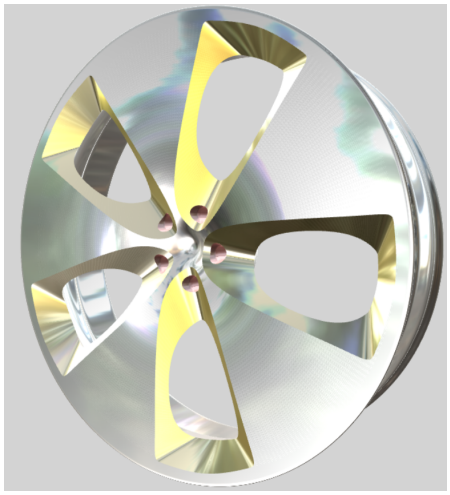

Figure 17: *Wheel 1: 11.4KBytes.*

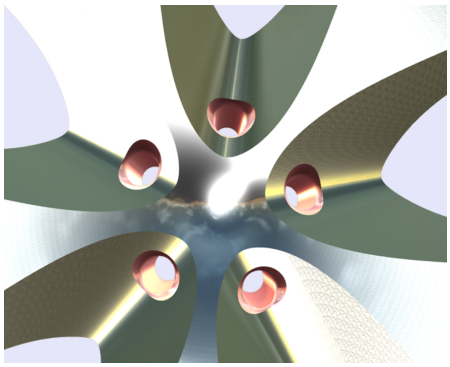

Figure 18: *Closeup of Fig. (17)*